Worksheet: Streams in Java 8 (and beyond)

Learning goals

Before the next day, you should have achieved the following learning goals:

- Use the stream api to process a list.
- Use streams to achieve *lazy* evaluation.
- Have seen examples of the basic functional operations: filter, map, reduce, etc.
- Compare parallel and serial execution using the stream api.

You should be able to finish most of non-star exercises in the lab session. Remember that star exercises are more difficult. Do not attempt star-exercises unless the other exercises are clear to you.

Lab Exercises

The following questions use the `Dish` class from the repository.

1. How would you use streams to filter the first two meat dishes?
2. How would you count the number of dishes in a stream using the `map` and `reduce` methods?

The next group of questions refer to a list of numbers.

3. (a) Given a list of numbers, how would you return a list of the square of each number? For example, given `[1, 2, 3, 4, 5]` you should return `[1, 4, 9, 16, 25]`.
   (b) Given two lists of numbers, how would you return all pairs of numbers? For example, given a list `[1, 2, 3]` and a list `[3, 4]` you should return

   `[(1, 3), (1, 4), (2, 3), (2, 4), (3, 3), (3, 4)]`

   For simplicity, you can represent a pair as an array with two elements.
4. How would you extend the previous example to return only pairs whose sum is divisible by 3? For example, `(2, 4)` and `(3, 3)` are valid.

All the remaining questions should be answered using the new Java 8 *Streams*. For all the exercises, start with a `List` of `String`s similar to this:

```
List<String> words = Arrays.asList("hi", "hello", ...);
```

5. (a) Loop through the words and print each on a separate line, with two spaces in front of each word.
   (b) Repeat the previous problem, but without the two spaces in front. This is trivial if you use the same approach as in (5a), so the point here is to use a method reference.

6. (a) For the following expressions, which you wrote for the last exercise sheet, produce the same transformations using `map`:
   - `List<String> excitingWords = transformedList(words, s -> s + "!");`
   - `List<String> eyeWords = transformedList(words, s -> s.replace("i", "eye"));`
   - `List<String> upperCaseWords = transformedList(words, String::toUpperCase);`

   (b) For the following lists produce the same transformations using `filter` (you wrote solutions for last exercise sheet):
   - `List<String> shortWords = allMatches(words, s -> s.length() < 4);`
   - `List<String> wordsWithB = allMatches(words, s -> s.contains("b"));`
   - `List<String> evenLengthWords = allMatches(words, s -> (s.length() % 2) == 0);`

7. (a) (*) Turn the strings in the array `words` into uppercase, keep only the ones that are shorter than 4 characters, and, of what is remaining, keep only the ones that contain `"e"`, and print the first result. Repeat the process, except checking for a `"q"` instead of an `"e"`.
   (b) (**) The above example uses lazy evaluation, but it is not easy to see that it is doing so. Make a variation of the above example that proves that it is doing lazy evaluation. The simplest way is to track which entries are turned into upper case.

8. (a) Produce a single `String` that is the result of concatenating the uppercase versions of all of the `String`s. E.g., the result should be `"HIHELLO..."`. Use a single `reduce` operation, without using `map`.
   (b) Produce the same `String` as above, but this time via a `map` operation that turns the words into upper case, followed by a `reduce` operation that concatenates them.
   (c) (*) Produce a `String` that is all the words concatenated together, but with commas in between. E.g., the result should be `"hi,hello,..."`. Note that there is no comma at the beginning, before `"hi"`, and also no comma at the end, after the last word.

9. (a) Write a `static` method that produces a `List` of a specified length of random numbers. E.g.:

   ```
   List<Double> nums = randomNumberList(someSize);

   // Result is something like [0.7096867136897776, 0.09894202723079482, ...]
   ```

   (b) Write a `static` method that produces a list of numbers that go in order by a step size. E.g.:

```
List<Integer> nums = orderedNumberList(50, 5, someSize);

// Result is [50, 55, 60, ...]
```

10. (a) (*) Provide three ways to use streams to compute the sum of a list of numbers.

    (b) (*) Rewrite one of the solutions for (10a) so that it can be executed in parallel; verify that you get the same answer as for the sequential code.

    (c) (**) Now, use streams to compute the product of some doubles. Show that serial and parallel versions **do not** always give the same answer.

    (Note: this is a bit tricky, because it seems at first that multiplication is associative, as required by the parallel `reduce`. It will be impossible to illustrate the result if you have a single-core computer.)